Process Management for Highly Parallel UNIX Systems

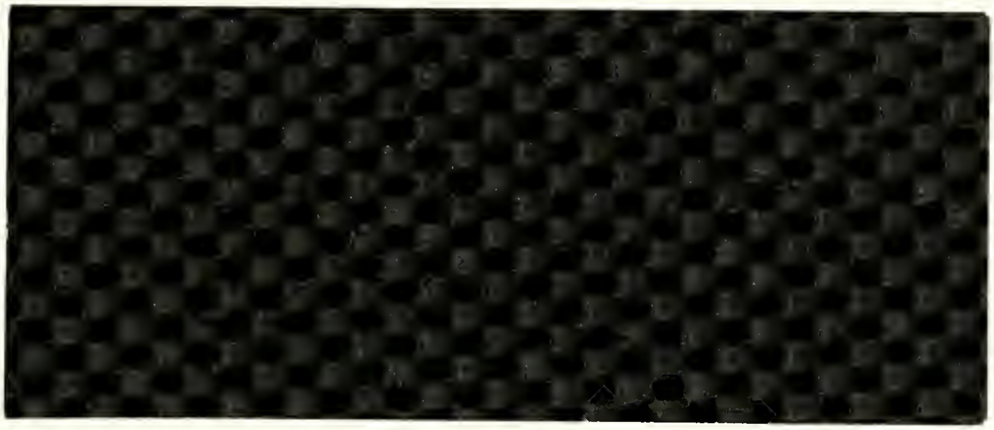
by

Jan Edler, Jim Lipkis, Edith Schonberg

Ultracomputer Note #136
April 19, 1988

Ultracomputer Research Laboratory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012



Process Management for Highly Parallel UNIX Systems

by

Jan Edler, Jim Lipkis, Edith Schonberg

Ultracomputer Note #136
April 19, 1988

This work was supported in part by I.B.M. under joint study agreement N00039-84-R-0605(Q), and in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under contract DE-FG02-88ER25052.

ABSTRACT

Despite early development exclusively on uniprocessors, a growing number of UNIX systems are now available for shared memory (MIMD) multiprocessors. While much of this trend has been driven by the general success of the UNIX interface as an emerging industry standard, experience has shown that the basic UNIX design is amenable to such environments. Relatively simple extensions such as shared memory and synchronization mechanisms suffice for many parallel programs.

While simple needs can be satisfied in a simple fashion, the desire to support more sophisticated applications has created pressure for ever more complex extensions. Is there a better way to meet such needs? Although some argue that it is time to abandon the UNIX model completely, we believe that viable alternatives exist within the traditional framework.

In this paper we propose several modifications to the process management facilities of the UNIX kernel. Some of them are primarily of interest for parallel processing, such as a generalized *fork* system call that can efficiently create many processes at once, while others are equally attractive in other contexts, such as mechanisms for improved I/O and IPC performance. While the primary goals are improved performance and reliability, a strong aesthetic judgement is applied to create a total design that is cohesively integrated.

While the concepts presented here are applicable to any UNIX environment, they have been conceived in the context of very large scale parallel computing, with hundreds or thousands of processors. An initial implementation of these extensions is currently underway for the NYU Ultracomputer prototype and the IBM RP3.

1. Introduction

A number of versions of the UNIX[1] operating system have been constructed for various shared memory (MIMD) parallel processors, with kernel interface semantics spanning a range from those with relatively minor extensions to those representing a radical departure from tradition. It is possible to support many parallel applications with only a single extension to the basic set of system calls, to provide for shared memory. In fact, a number of versions of UNIX for uniprocessors, such as System V [AT&T86], have had this feature for years. Unfortunately, the range of parallel applications that can be efficiently supported on such a minimally extended system is rather narrow, leading to an assortment of further extensions, such as:

- Various non-busy-waiting synchronization mechanisms, e.g. locks, semaphores, events, etc.
- Different kinds of processes with more efficient semantics than standard processes.
- New representations of process state for executing, debugging, core-dumping, and/or checkpointing parallel programs.
- Enhancements to the scheduler to improve performance of parallel jobs.
- Enhancements to I/O and IPC performance.

With demand for new extensions like these increasing the complexity of the system, the idea of adopting a new system model and starting with a fresh design (free from the burden of existing UNIX semantics) seems like an attractive alternative. Unfortunately, a completely new design loses many of the *benefits* of UNIX as well, including compatibility and familiarity. Adopting UNIX compatibility as a constraint on a fresh design simply leads to *even more* complexity, from the inevitable proliferation of similar-but-different mechanisms and numerous "special cases" required to hold the whole thing together.

[1] UNIX is a registered trademark of AT&T.

In this paper we discuss a set of modifications, primarily in the area of process management, that we believe form a natural generalization and build upon the strength of existing UNIX facilities, in order to provide efficient support for sophisticated parallel applications in a system with manageable complexity. Although they differ substantially from the old facilities in some respects, compatibility and implementability have been carefully considered. Despite the fact that these proposals have been conceived and developed for highly parallel machines (with potentially thousands of processors), such as the NYU Ultracomputer [Gottli87] and the IBM RP3 [PBG85], we have tried not to lose sight of smaller machines and uniprocessors as well.

The proposals in this paper are generally made in the context of the Symunix II operating system, currently being designed and implemented at NYU. We believe that they form a cohesive whole when taken together with the memory management interface described in [ELS88].

We begin the next section by considering mechanisms to control parallelism, that is, how to create and manage multiple instruction streams, and discuss a spectrum of implementation models which the new design can support. This leads into the following section where control of processor scheduling is addressed. In section 4, asynchronous system calls, important for the support of highly-efficient user-mode multitasking as well as high performance I/O, are introduced. We then go on in section 5 to address synchronization and IPC issues, and extend the signal mechanism in section 6. We conclude with a summary and closing arguments in section 7.

2. Parallelism

The first question that must be asked when considering MIMD processing in a UNIX environment is what form do the multiple instruction streams take, how are they created, destroyed, and managed?

2.1. Processes

Given some form of shared memory, an acceptable environment for many parallel applications can be constructed with the established **fork**, **wait**, and **exit** system calls, where each instruction stream is represented by a UNIX process. While this approach is relatively easy to implement, there are several reasons why it is inadequate for any but the simplest of environments:

- Since processes must be created and waited-for one at a time, **fork** and **wait** represent a potential serial bottleneck. Whether or not this bottleneck is severe depends on the number of processes to be created and the proportion of the total run time of an application that is consumed by **fork** and **wait**. As an improvement to serially process initiation, a logarithmic approach may be used in which each new process helps create more processes, but this implies more complexity and overhead, and has another problem discussed in the next bullet:
- Processes created by **fork** form a hierarchy, which can be awkward within a single parallel application where no such hierarchical relationship logically exists. The number of processes assigned to an application may grow and shrink over time, not according to block structured nesting, but according to the amount of parallelism in an application during various phases of execution and the availability of processors on the machine. This would not be a problem except for the way process termination is reported to the parent (or **init**, if the parent has already terminated) via the **wait** system call and an optional signal.
- There is insufficient control over the kinds of state changes which cause **wait** to return information. For example, it is not possible to ignore normal terminations and wait only for abnormal ones.

In addition, it is obvious that **fork** involves far too much overhead (even with the best of implementations) to be used to create the multiple instruction streams of parallel programs, in particular those with fine-grained parallelism. Because of this, many designers have questioned the very nature of the UNIX "process" abstraction, and its suitability as a vehicle for parallel applications. There are a variety of proposals, including MACH's threads [Tevani87], UNICOS's tfork operation [Reinha85], and others [Kepecs85], but here we will simply refer to any of them as *lightweight processes*, because a primary goal is to reduce the "weight" of processes by removing some attributes such as private address space, open files, signals, current directories, etc., normally necessary to support UNIX semantics. All of this "unnecessary weight" contributes to the overhead of process creation, context-switching, etc.

There are three common misconceptions about lightweight processes that we wish to address:

Simplicity

It is often claimed that simplicity is gained by shedding a multitude of unnecessary features associated with the UNIX process. This would probably be true except for the fact that most of these features are *desirable*, making it necessary to re-implement them in some other way, either in the kernel or in some combination of libraries and server processes. In either case, it is quite possible that the overall result (once the discarded UNIX semantics are re-implemented) will add complexity to the implementation rather than subtract from it. The size and complexity of the programmer's interface will also increase, with the addition of lightweight process semantics. For example, elimination of the private address space is incompatible with established UNIX semantics, so most proposals simply add an optional shared address space concept to the system: an additional kernel feature certainly doesn't *reduce* overall complexity.

Efficiency

Lightweight processes can be created and destroyed more quickly than heavier traditional processes. However, no form of kernel-supported process (light or heavy) will be efficient enough to be created and destroyed on demand in fine-grained parallel applications. And in sufficiently coarse-grained applications the creation/destruction expense will not be an issue. There are no data yet on how many programs fall "in between" these two extremes, but in almost all cases the most efficient way to implement parallel applications is to create processes (light or heavy) in advance of need and schedule application-defined instruction streams to those processes without kernel intervention. Context-switching between lightweight processes that are part of the same application can be done more efficiently than between heavyweight processes, but it can be done even faster without kernel intervention. Implementation of kernel-supported lightweight processes would therefore represent an unnecessary duplication of function.

Demand Paging

When virtual memory techniques such as demand paging are a requirement, it may seem that lightweight processes implemented outside the kernel are not viable: if each instruction stream is a separate process, the kernel can schedule another one when a page fault occurs, but if streams are scheduled to processes in user-mode (as advocated above) that isn't possible. The same situation would arise if page faults were handled by hardware suspension of the faulting processor, without an interrupt to trigger a reschedule. It is possible to use the same solution outside the kernel: a signal can be delivered to notify a process of certain paging events (e.g. page fault, page input complete, page eviction). Provided certain pages are "locked" in memory (to prevent recursive page faults in signal handling and instruction stream management), then the process can switch to another stream. Assuming a moderate page fault rate, the performance gain from scheduling outside the kernel will outweigh any additional overhead from signal handling.

For these reasons we believe lightweight processes offer insufficient advantages to justify their addition to the system as a basic kernel-supported facility. Instead, we extend the traditional process model, and rely somewhat more upon standard library and language runtime support software than most previous UNIX kernels, to provide superior performance outside the kernel.

2.2. The Process Control Bottleneck

Our first extension is conceptually simple: we modify **fork** to create more than one new process at a time. The enhanced **fork** is called **spawn**, and looks like this:

spawn (**int** n, **int** flags, **int** *pids)

The parameter *n* tells how many new processes to create, and *pids* points to a suitably large integer array, where the process ID of each child is stored. The return value is zero in the parent and a unique *spawn index* chosen from the set {1,...,*n*} in each child, such that, for each child *i*,

$pids[\text{spawn\_index}_i - 1] = \text{pid}_i$.

We define the term *family* to refer to a set of processes related by **spawn** but not **exec**; i.e. a single parallel program, with the various processes presumably cooperating. The original member of the family (the only one that has ever done an **exec**) is known as the *progenitor*. The process hierarchy awkwardness described in section 1.1 is addressed by providing a **spawn** option (a flag bit) that makes the progenitor the effective parent of all processes in the family, thus collapsing the hierarchy. If the option is set, attributes such as address space[2], open files, etc. are derived from the true parent, but SIGCHLD signals generated by the children are sent to the progenitor. The call **getppid** in the children will return the process ID of the progenitor.

The notion of progenitor ensures that child process status will not be lost if a parent process terminates – provided, of course, that the progenitor itself doesn't terminate. The addition of a new signal type, SIGPARENT, with default action "ignore", allows children to be notified in case of unexpected death of their parent (or progenitor). This is important for parallel applications that synchronize, since a process terminating unexpectedly (perhaps due to a program bug, such as division by zero) without notification to the rest of the family can deadlock the remaining processes.

What happens to orphaned children of parents and progenitors when SIGPARENT is ignored? In UNIX, when a parent terminates without waiting for all of its children to terminate, they are "inherited" by the **init** process. This is a slight convenience to some UNIX kernels, but we are unaware of any version of **init** that actually depends upon this behavior. Therefore, we have chosen to simply not send SIGCHLD for orphaned processes.

Rather than generalizing the **wait** system call with new options to specify just which processes to wait for, we have chosen to eliminate **wait** in favor of an enhanced SIGCHLD signal. We defer a discussion of other signal-related issues to section 6, but for now the reader should assume that SIGCHLD signals will be sent to the parent for each "interesting" state change of a child (e.g. termination), that these signals are queued so that none are lost, and that SIGCHLD is accompanied by additional data passed as extra arguments to the parent's signal handler, providing essentially the same information as that obtained by **wait**.

Given the above, it is not hard to build compatible versions of **fork**, **wait**, and a SIGCHLD handler that maintain a list of processes not yet waited-for. The only non-obvious aspects of the job are to properly handle the case where the parent calls **exec** before waiting for all its children, and to support old software that uses SIGCHLD with the old semantics, but it can be done.

So what advantage does this scheme have, other than eliminating one system call? The parent can specify exactly which kinds of state changes are of interest by or'ing together certain flag values when calling **spawn**, and the type of each state change is included in the extra arguments passed to the signal handler. The following state changes can be selected:

Termination by signal
Termination by **exit**(0)
Termination by non-zero **exit**
Stopped by signal
Continued by signal

When a child changes to a state selected as interesting by its parent, SIGCHLD with appropriate arguments will be sent. The role of the zombie process is replaced by the queued signal mechanism. Most parallel programs will probably only be interested in abnormal terminations (termination by signal and non-zero **exit**).

[2] Shared memory is provided by allowing shared memory segments to be inherited on spawn and also through file mapping, as described in [ELS88].

2.3. Implementation Models

Given a fixed kernel interface, the implementor of parallel language environments is still faced with a large number of options, or tradeoffs, with significant impact on the functionality and efficiency of the resulting environment. At one extreme, the kernel interface described in this paper is available to the experienced C programmer, and can be used directly. While such an "environment" is extremely flexible, it is also very primitive. At the other extreme, languages with explicit parallel constructs require implementations with runtime support software to map the language-supported abstractions into the kernel-provided ones.

As has already been mentioned, using **fork** and **spawn** directly as the method of creating new instruction streams is too expensive for all but the most static types of parallel applications. It is more efficient for language-specific runtime support software (executing in user-mode) to schedule application-specific instruction streams to processes that have been created in advance; we refer to such processes as *prespawned*, and to the instruction streams as *tasks*. If tasks don't run to completion, but *block* during synchronization to allow other tasks to run on the same process, we say the runtime support system is *multitasking*.

A tasking system that schedules prespawned processes duplicates certain operating system functions, such as context switching, task creation/destruction, etc. However, depending on the task semantics being supported, the user-mode system can be orders of magnitude more efficient than an operating system. The efficiency/generality tradeoffs of tasking systems are illustrated by the following:

- If no task ever waits for another, then the tasking system can use an extremely efficient run-until-completion scheduling paradigm. For many scientific applications written in Fortran extended with a parallel loop construct, such a simple scheduling paradigm is adequate.
- If tasks not only block, but must be *preempted*, a much more sophisticated multitasking system is necessary. The Ada[3] language, for example, requires a scheduler that recognizes priorities and preemption.
- If tasks never make requests for I/O or other system services, then the multitasking system can be simpler and more efficient.
- One might imagine supporting full UNIX process semantics in user-mode tasks, but this is difficult, expensive, and not generally useful. For example, signals with full UNIX semantics require process IDs, which would be hard to simulate in the fullest generality for tasks if they must interact with real (e.g. unrelated) processes. Operations which manipulate the address space in an incompatible way (e.g. **exec**) would be pointless, because context-switching in user-mode between tasks with incompatible memory images would be even more expensive than context-switching in the kernel.

It is important to note that, although the choice of semantics supported by a particular parallel language environment will have a profound effect upon the complexity and efficiency of the language runtime implementation, the nature of the underlying kernel interface will have much less effect. Techniques such as prespawning and multitasking remain important regardless of whether or not the kernel supports lightweight processes, or the way in which important system components (e.g. the file system, virtual memory system, network protocols, etc.) are implemented (e.g. inside or outside the kernel).

3. Scheduling

Scheduling has traditionally been transparent in UNIX, that is, with the exception of the **nice** system call (**setpriority** in 4.2BSD), there have been essentially no scheduling controls available to the programmer or system administrator. Scheduling is performed based on priorities that are adjusted dynamically according to resource usage and system load. In general, that is good; unnecessary controls on a system should be avoided. However, with the presence of parallel applications, new factors emerge that make some change increasingly necessary:

[3] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

- *Frequent synchronization between processes.* From the point of view of a single parallel application running on a multiprocessor, the most efficient way to perform such synchronization is to run all synchronizing processes simultaneously, and allow them to *busy-wait* as necessary during the synchronization. Most parallel architectures directly support an efficient hardware mechanism to do this, avoiding the overhead associated with blocking a process and context switching to another one. Unfortunately, even on a multiprocessor there aren't always enough processors, so some form of process preemption is often still needed. However, busy-waiting in the presence of preemption can be very bad for performance. For example, a process might busy-wait a substantial amount of time for another process that has been preempted, creating a feedback effect by performing useless work that effectively raises the system load. This problem arises because the scheduler isn't aware that synchronization is going on.
- *Uneven rates of execution.* Some parallel algorithms perform poorly when the instruction streams don't progress at approximately the same rate. While even progress is a property of any "fair" scheduler over a sufficiently long period of time, some algorithms have less tolerance and require fairness over shorter time periods. An example is an algorithm that uses *barrier synchronization* [Jordan78] frequently. In such a situation, the progress of the overall algorithm is limited by the progress of the last process to reach each barrier.
- *Self scheduling.* Beyond the problems we have just described, where lack of control over preemption can cause serious performance problems, there are applications that ask almost nothing of the operating system at all, except that it not interfere. What is often wanted by the user for these applications is a long-term commitment of resources (e.g. processors, memory), and, given that they are willing to pay the price for such commitment, they expect to get performance near the absolute maximum deliverable by the hardware, with no operating system overhead. Such control may not be appropriate on small machines or uniprocessors, but becomes increasingly important on large high-performance multiprocessors.

The common theme running through all these problems is preemptive scheduling of processes, and we now propose two new mechanisms to eliminate or control it: *scheduling groups* and *temporary non-preemption.*

3.1. Scheduling Groups

We introduce a new process aggregate, the *scheduling group,* which is identified by a *scheduling group ID.* Every process is a member of a scheduling group, although that group may consist of only the single process itself. In many cases, we expect scheduling groups to correspond to families, as defined in section 2. A scheduling group is subject to one of the following scheduling policies:

fully-preemptable
This is the traditional scheduling policy of UNIX, and is the default.

group-preemptable
The members may only be preempted or rescheduled at (approximately) the same time.

non-preemptable
The members are immune from normal preemption.

Our notion of group-preemption is similar to Ousterhout's coscheduling of task forces [Ouster82], but is concerned with preemption rather than blocking.

Since it is necessary for all members of a group-preemptable or non-preemptable group to execute simultaneously, it must be possible to meet their combined resource requirements. It is obvious that uniprocessors can't provide these new scheduling policies for groups containing more than one process, but multiprocessors also have a finite supply of processors and memory, and since spawning a new process into such a group or requesting more memory are operations that increase the resource requirements of the group, it is possible for them to fail. To avoid this problem, we must provide for long term allocation of processors and memory. Long term resource allocations, group-preemption, and non-preemption are restricted by system administrative policy and implemented by a long-term component of the scheduler. In addition to new system calls for manipulation of scheduling group IDs, new calls are needed to select the scheduling policy of a group, and request long-term allocation of processors and memory.

These new mechanisms directly support multiprocessor users whose primary objective is to "get rid" of operating system overhead, and solve the other problems mentioned above, provided that the users are willing to pay the cost which may include restrictions on resource utilization, increased average turnaround time, and in some cases a negative impact on interactive or real-time responsiveness.

The group-preemptable and non-preemptable scheduling policies have no effect on voluntary blockages (e.g. when a process issues a system call for which it is necessary to block). When a process is blocked, other members of the group will continue to be run, preempted, and rescheduled according to the current policy. Note also that the scheduling policy does not necessarily affect interrupt processing, so a process may still suffer the overhead of interrupts caused by some other process doing I/O; this is an unavoidable overhead of multiuser activity on many architectures.

3.2. Temporary Non-Preemption

Group- and non-preemptable scheduling policies are conservative mechanisms for users who need long-term control over preemption; as such they are expensive because of the long-term commitments required. There are other situations, in which synchronization occurs for short periods of time, that can be satisfactorily handled by a cheaper mechanism; we introduce the notion of *temporary non-preemption* for this purpose. This is a "hook" into the scheduler to allow processes to accommodate very short-term conditions. An example of typical usage is to prevent preemption while holding a short-duration busy-waiting lock. The process essentially says, "I don't mind being preempted, but please wait a bit until I tell you it's ok". The mechanism provides protection against malicious processes that never say "ok". Unlike the scheduling policies described above, it is useful on uniprocessors as well as multiprocessors. Two calls are provided, **tempnopreempt** and **tempokpreempt**. The first notifies the scheduler that temporary abeyance of preemption is desired, and the second signals a return to normal scheduling.

Much of the usefulness of this feature derives from the fact that the implementation is extremely efficient. The scheduler maintains for each process the address of a two word vector in the user address space; this address is set via a new system call normally issued by the language start-up code immediately after **exec**. The first word is for communication from the user to the scheduler, the second is for communication from the scheduler to the user. Initially, both words are set to zero. What **tempnopreempt** does is essentially

```
word1++;
```

while **tempokpreempt** is approximately

```
if (--word1 == 0 && (temp=word2) != 0) {
        word2 = 0;
        yield();
        return temp;
}
return 0;
```

Yield is a new system call that reschedules the processor. Because *word1* is incremented and decremented, rather than simply set, **tempnopreempt** and **tempokpreempt** can be safely nested. What the scheduler does is complementary; when it wants to preempt a process, code such as the following is executed on the same processor (i.e. the scheduler runs as an interrupt):

```
if (word1 == 0)
        ok to preempt;
else if (preemption not already pending for this process) {
        word2 = 1;
        note preemption pending;
}
else if (preemption pending for at least tlim time) {
        word2 = 2;
        ok to force preemption;
}
```

Where *tlim* is perhaps a few milliseconds, and the *preemption pending* state is maintained per-process and cleared on actual preemptions and on **yields**. The purpose of the **tempokpreempt** return value is to notify the user (after the fact) if preemption was requested or forced.

Overhead for this implementation is only a few instructions in the common case where no preemption would have occurred anyway, and only the overhead of the **yield** otherwise. The most abusive a user can get with this scheme is to lengthen a time-slice by *tlim*; it is easy to prevent any net gain from such a strategy by shortening the next slice by *tlim* to compensate.

4. Asynchronous System Calls

The basic UNIX I/O model of open/close/read/write/seek operations on byte streams is simple, powerful, and quite appropriate for the workload to which UNIX systems have traditionally been applied, but in higher performance environments some extensions are often necessary. One such extension is asynchronous I/O, which has existed in non-UNIX systems for decades and even within the UNIX kernel since its inception, but only fairly recently has it begun to become directly available to the user, on systems such as the Cray [Cray86] and Convex [Convex87]. Of course it is possible to obtain asynchronous I/O on any UNIX system that has shared memory, simply by using another process to issue the synchronous I/O call. Our objections to such an approach are that it is less efficient since additional context switches are required, and it is more complex for the user software.

In large-scale parallel systems, user access to asynchronous I/O is desirable not only for high bandwidth and increased overlapping of processing and I/O activity, but also to facilitate the construction of efficient user-mode multitasking systems, as described in section 2. In such an environment, when a task does a synchronous read or write system call, the process executing the task can become blocked, thus "wasting" a process or processor if other tasks are ready to run; a better approach is to use asynchronous I/O, either directly, in the application, or indirectly, in runtime support software implementing the standard synchronous I/O model for tasks.

4.1. Meta System Calls

Rather than propose a special-purpose extension for certain favorable cases of I/O, we propose a general strategy that can be applied to any system call for which asynchronous operation makes sense, e.g. not only **read**, **write**, **open**, and **close**, but also **ioctl**, **mkdir**, **rmdir**, **rename**, **link**, **unlink**, **stat**, and so forth. The general idea is to define a control block for asynchronous system calls, identifying the specific call and giving the arguments. Additional fields are provided within the control block for returned values and status (e.g. pending, successfully completed, or completed with an error). Three new *meta system calls* are introduced, each taking a control block pointer as an argument:

syscall
: Issue a system call. The return value indicates whether or not the call has already completed, if not a signal (SIGSCALL) will be delivered when it is.

syswait
: Wait for completion of a system call. The return value distinguishes between control blocks that aren't pending, those that have just completed, and those that have been cancelled.

syscancel
Cancel a pending system call, if possible.

The **syscall/syswait/syscancel** framework relieves the programmer from having to know exactly which calls are asynchronous and which aren't. It is even possible for a given call to sometimes complete synchronously and sometimes asynchronously; the **syscall** return value lets the user know.

In some cases, new calls or modified semantics must be introduced to make the most of asynchrony. For example, versions of **read** and **write** can be provided with explicit file position arguments, eliminating the need for separate **lseek** calls. This is important for asynchronous I/O since operations can be completed in any order.

Asynchronous system calls require queued signals with arguments. When SIGSCALL is delivered, the signal handler is called with an extra argument, a pointer to the completed control block. Instead of requiring all uses of asynchronous system calls within a program to use the same signal handler, we allow the user to specify a different handler in each control block.

4.2. Higher Level Interfaces

As a matter of convenience, we expect the development of a higher level interface to the asynchronous system calls along the lines of

aread (**int** fd, **void** *buf, **ulong** count, **ulong** pos, **void** (*func)());

where the function pointed to by the formal argument *func* is called on completion with a pointer to the actual control block, so that the various fields may be inspected as necessary. The implementation of such a higher level interface is merely a matter of allocating the control block, issuing the **syscall**, and after the completion signal is delivered, calling (**func*) and finally deallocating the control block.

In V7 and similar UNIX implementations, when a signal arrives during a "slow" system call (such as **wait**, **pause**, or some cases of **read** or **write** on "slow devices"), and a handler has been set for the signal, the system call is interrupted. If the handler returns, things are arranged so that the interrupted system call appears to return a special error, EINTR. Because it is usually inconvenient to deal with EINTR, systems compatible with Berkeley UNIX will automatically restart the system call in some cases. The introduction of asynchronous versions of all interruptible system calls allows us to avoid the problem altogether, by simply not interrupting the system call. If the handler really wants to interrupt a system call, it can do so by invoking **syscancel**. Implementation of the standard synchronous interface calls as a higher level interface above **syscall**, while not required, is fairly straightforward although there is some impact in machine-dependent signal-support code, such as the *trampoline* code and *longjmp*.

5. Synchronization and IPC

Shared memory multiprocessor architectures support a variety of underlying synchronization mechanisms. In addition to the common test-and-set and compare-and-swap primitives, these include tagged-memory schemes (e.g. full/empty bits) [Smith81] as well as generalized atomic read-modify-write operations such as fetch&add [Gottli87, PBG85]. Parallel programs, on the other hand, employ a wide variety of high-level synchronization functions. For maximum efficiency it is necessary to implement these directly in terms of the available hardware features. For example, simple mailboxes are well served by full/empty tags, while counting semaphores, barrier synchronization, and readers/writers locks (among others) are simply and efficiently implemented with fetch&add; of course, any of these architectural mechanisms can support simple mutual exclusion. These and other coordination functions can be implemented with minimal expense by taking advantage of the fact that in the common cases where no contention or need for delay exists, only a small number of instructions are required.

5.1. Process Management for Synchronization Functions

We believe that semaphores and other synchronization facilities should be provided in library and language runtime code. Kernel implementations (e.g., semaphores in System V) [AT&T86] will necessarily limit flexibility and discourage efficient exploitation of available hardware features. The role of the kernel should be limited to the management of process state, while communication and coordination among processes are implemented in user-mode. On multiprocessors, both busy-waiting and process-switching modes of synchronization are needed; the former to minimize the overhead of extra context switches, the latter to minimize idle processor time. A hybrid of busy-waiting and process-switching (also known as "two-phase blocking") [Ouster82] can combine the performance advantages of the two modes. Busy-waiting requires no support from the operating system, though the group scheduling policies of section 3.1 and the temporary non-preemption facilities discussed in Section 3.2 are useful for avoiding performance problems that can occur when process scheduling interacts with synchronization. Process-switching, on the other hand, requires system services for process suspension and resumption that are not subject to race conditions.

Several solutions have been proposed to this problem. The key to our mechanism is a per-process *pending unblock* flag maintained by the kernel as part of the process state. A new **block** system call atomically tests the *pending unblock* flag, and, if set, clears the flag and returns immediately; otherwise, it suspends the process. The **unblock** system call atomically tests whether the indicated process was suspended by a **block**, and, if so, makes it ready; otherwise it sets the *pending unblock* flag for the process. Reliable (i.e., race free) synchronization routines are possible because an **unblock** is effective even when it happens to occur before the target process has issued the corresponding **block**. For efficiency, **block** and **unblock** are avoided completely in the most common contention-free case. Furthermore, no other locking is needed when the primitive hardware-supported synchronization facility is suitable. This is important for cases like counting semaphores and readers/writers locks where it is possible to completely avoid serialization as long as blocking is not logically required.

MACH provides a different but equivalent mechanism for reliable suspension and resumption of threads, though it requires three system calls rather than two [Tevani87]. In UNIX, the same function can in fact be provided using the standard signal mechanism (**pause** and **kill** system calls). However, there are efficiency and operational drawbacks to this approach, including the requirement for a new, dedicated signal type.

5.2. Low Overhead IPC

Message passing remains a useful paradigm on shared memory architectures. Systems such as Accent [Fitzge86] have demonstrated the efficiency to be gained by utilizing copy-on-write techniques when passing large messages. We would like to exploit memory sharing to provide very low overhead communication channels for small messages as well. Stream-oriented IPC applications can be implemented in the runtime library so as to execute almost entirely without kernel intervention once shared buffers are established. Producer/consumer coordination, using the above synchronization facilities, ensures correct operation at very low cost.

A user-mode implementation of UNIX pipes seems particularly straightforward because communicating processes necessarily share a common parent. Memory regions that survive **exec**, as supported by the Symunix II memory management system [ELS88], can be allocated by the **pipe** "system call" for use as shared communication buffers. Library code (primarily in the **open**, **close**, **read**, and **write** routines) then implements the standard pipe semantics. The only remaining difficulty is support for the SIGPIPE signal, which could be done with a slightly more general per-open-file flag to distinguish logical "readers" from "writers" and generate a signal when the last "reader" maps out or closes the file.

6. Signals

Since the main focus of our work has been on shared memory MIMD machines, we regard signals as playing a supporting rather than a central role in the system. That is, we are not primarily concerned with the use of signals for general IPC, but as a mechanism to support other facilities. Nevertheless, when modifications are required, we seek general solutions.

6.1. Signal Queuing

In order to adequately support enhanced child state-change notification (discussed in section 2) and asynchronous system calls (discussed in section 4), we need to queue pending signals that are to be caught. This means that all signals sent will be separately received with no omissions or duplications, and signals of a given type will be received in the order sent. Counting pending signals would be simpler, but doesn't allow additional data to accompany the signal (e.g. child status or pointer to control block). While it is not necessary that queuing be introduced for other signal types, it seems worthwhile to have similar behavior for all signals (and we doubt the existence of many applications depending on non-queued behavior). The only signals for which queueing makes no sense at all are signals that aren't catchable or maskable, such as SIGKILL and SIGSTOP. Also, certain signals sent by the kernel in response to synchronous events (e.g. SIGILL, SIGFPE, SIGSEGV), probably should not be maskable (or sendable by system call) and therefore might as well not be queued.

Because of signal queuing, a new resource type (a "signal delivery structure") must be introduced into the kernel, and it is possible (although unlikely) for this resource to be depleted, preventing a signal from being sent. A partial solution to this problem is to pre-reserve delivery structures as soon as the possibility of future need arises. For example, when a process is created, structures for SIGCHLD and tty-generated signals can be reserved. This approach can be used together with other techniques such as multiplicity counts (to compactly represent many consecutive identical signals) and possibly a limit on the length of the per-process pending signal queues.

6.2. The Word Size Barrier

Extending the number of signal types to more than the number of bits in an **int** causes problems with **sigblock**, **sigpause**, and **sigsetmask**; 4.3BSD already defines 31 signals, and we have already identified two new needed signals, SIGPARENT and SIGSCALL, in this paper as well as a new signal for checkpointing, SIGCKPT in [ELS88]. One solution to the problem is to extend the signal interface by adding new routines like **lsigsetmask** that take a pointer to a bitstring. Compatibility is only a minor issue because the extended interface is only required when the new signals are used. While this solution appears adequate to support the addition of a few new signal types, the problem seems general enough to be worth a more complete solution. We are considering the implementation of per-signal actions in a single language-specific first-stage signal handler instead of the kernel; this would allow a small number of maskable *levels*, but a nearly unbounded number of signal *types*.

6.3. Efficient Signal Masking

In uniprocessor operating systems, critical sections can be implemented by masking interrupts, but multiprocessors require some form of locking in addition. The same situation exists for serial and parallel applications, but with signals taking the place of hardware interrupts. When using signal masking and locking together, the overall performance depends on the efficiency of both mechanisms. Since we have identified low-overhead locking schemes, it remains to describe a complementary method of signal masking.

We propose to implement the signal masking functions without system calls, in a manner analogous to the implementation of temporary non-preemption described in section 3. In this case, the address of a single word (*sigword*) and two bitstrings in the user address space are maintained by the kernel. The user can set *sigword* to a non-zero value to mask all maskable signals, and *bitstring1* to mask only certain signals. The kernel examines *sigword* and *bitstring1*, and modifies *bitstring2* to notify the user of pending masked signals. The implementation of **sigblock** is essentially

```
        sigword++;
        oldmask = bitstring1;
        bitstring1 = oldmask | newmask;
        sigword--;
        if (bitstring2 & ~bitstring1)
                sigcheck();
        return oldmask;
```

where *newmask* is the argument and **sigcheck** is a new system call that causes all non-masked pending signals to be handled. Whenever a signal is sent to a process, the kernel examines *sigword* and *bitstring1* to see if it is masked or not, and it adjusts *bitstring2* if necessary whenever a signal is sent or handled. The use of *sigword* is not necessary on machines that support fetch&or, fetch&and, and fetch&store [Gottli81] on objects the size of the bitstrings.

7. Summary

We have presented a collection of proposals for enhancing the ability of UNIX systems to support sophisticated parallel applications on shared memory MIMD computers. It is possible in some cases to realize the advantages of an isolated subset of the proposed mechanisms, such as the scheduling controls discussed in section 3, but we believe that much of their strength comes from their use in combination, together with the memory management system of [ELS88]. Common themes include reliance on user-mode libraries and runtime support software for performance-critical functions, and extension of UNIX facilities to meet high-performance requirements.

Acknowledgements

The ideas described in this paper have benefited from many discussions with other members of the Ultracomputer Research Laboratory at NYU, and also with researchers on the IBM RP3 project.

References

[AT&T86]
AT&T, *System V Interface Definition, Issue 2,* AT&T (1986).

[Convex87]
Convex Computer Corporation, *Convex UNIX Programmer's Manual, Verstion 6.0.* Nov. 1, 1987.

[Cray86]
Cray Research, Inc., *UNICOS System Calls Reference Manual (SR-2012).* March, 1986.

[ELS88]
Jan Edler, Jim Lipkis, and Edith Schonberg, "Memory Management in Symunix II: A Design for Large-Scale Shared Memory Multiprocessors," Ultracomputer Note 135, New York University (April 1988).

[Fitzge86]
Robert Fitzgerald and Richard F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems* 4(2) pp. 147-177 (May, 1986).

[Gottli87]
Allan Gottlieb, "An Overview of the NYU Ultracomputer Project," pp. 25-95 in *Experimental Parallel Computing Architectures*, ed. Jack J. Dongarra, North Holland (1987).

[Gottli81]
Allan Gottlieb and Clyde Kruskal, "Coordinating Parallel Processors: A Partial Unification," *ACM Computer Architecture News*, (Oct., 1981).

[Jordan78]
Harry F. Jordan, "Special Purpose Architecture for Finite Element Analysis," *Proc. 1978*

International Conference on Parallel Processing, pp. 263-266 (August, 1978).

[Kepecs85]
Jonathan Kepecs, "Lightweight Processes for UNIX Implementation and Applications," *Proc. Summer USENIX Conference*, pp. 299-308 (June, 1985).

[Ouster82]
John K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. 3rd International Conf. Distributed Computing Systems*, pp. 22-30 (1982).

[PBG85]
G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 International Conference on Parallel Processing*, pp. 764-771 (Aug. 1985).

[Reinha85]
S. Reinhardt, "A Data-Flow Approach to Multitasking on CRAY X-MP Computers," *Proc. 10th ACM Symp. Operating Systems Principles.*, pp. 107-114 (Dec., 1985).

[Smith81]
Burton J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Real Time Signal Processing IV, Proceedings of SPIE*, pp. 241-248 (1981). reprinted in *Supercomputers: Design and Applications*, ed. Kai Hwang, IEEE Computer Society Press, 1984.

[Tevani87]
Avadis Tevanian, Jr., Richard F. Rashid, David B. Golob, David L. Black, Eric Cooper, and Michael W. Young, "MACH Threads and the UNIX Kernel: The Battle for Control," *Proc. USENIX Conf.*, (June, 1987).

NYU UCN-136
Edler, Jan
Process management for
 highly parallel UNIX
 systems. c.1

DATE DUE	BORROWER'S NAME
